Classes

definition, creation, etc.

Definition vs. Creation

The definition of a class is something that should already be familiar to you at this point, as all Java programs take place within the context of a class. For instance:

```
// The definition of a class called: A

class A {
    int i;
    boolean b;
    String s;
}
```

defines a class called `A` which has three member variables (`int i`, `boolean b`, `String s`) and no methods. Notice that since this class has no methods it cannot be used from the command-line. As we know, a class must have a `main()` method in order to initiate a program.

The definition of the class `A`, in and of itself, does not create any objects that are instances of `A`. The two operations are separate, though related. First, we define a class – in other words, give a description of a data type. Then, we can create objects that are instances of that class. Consider a separate class: `User`.

```
class User {
    public static void main(String [] args) {
        A aObj = new A();                  // creates an object of type: A

        System.out.println(aObj.i);        // prints 0
        System.out.println(aObj.b);        // prints false
        System.out.println(aObj.s);        // prints null
        return;
    }
}
```

The first class defined a data type called: `A`. The second class was the definition of a type called: `User`. The `User` class contains a `main()` method that uses the definition of class `A` to create an object of type `A`. The creation of an `A` object is indicated in the following line:

```
A aObj = new A();
```

The proceeding lines print the values of the member variables of `aObj`. Even though none of the variables were given values explicitly, the creation of any object provides default values to that object's member variables (keep in mind that this is not the case with variables that are local to a function):

Numeric	:	0
Boolean	:	false
Refernces	:	null

Notice that in the definition of `main()` in the `User` class, no `User` objects have actually been created. This is because *calling the* `main()` *method of a class does not, in and of itself, create* *an object of that class!* That being said, the `main()` method of a class *may* create an object that class. A `main()` method could be written into class `A`, for instance:

```
// The definition of A with a main() method

class A {
    int i;
    boolean b;
    String s;

    public static void main(String [] args) {
         A aObj = new A();                    // creates an object of type: A

        System.out.println(aObj.i);         // prints 0
        System.out.println(aObj.b);         // prints false
        System.out.println(aObj.s);         // prints null
        return;
    }
}
```

The `main()` method above declares an object of class `A`, and then prints the values of its member variables. It may be peculiar at first to think about calling a method of a class without creating an object of that class, however this notion will become intuitive with time. Furthermore, an explanation of the `static` keyword will make things clearer.

Constructors

The purpose of a constructor is to initialize an object and ensure that it is in a valid state. Qualifying as a valid state depends on how each class is defined and intended to be used, so this can vary dramatically across classes. Nonetheless, each class is given a ***default constructor***, if the writer of the class does not explicitly write one. For instance, consider the following `Triangle` class:

```
class Triangle {
    int angle1, angle2, angle3;
}
```

An object of the `Triangle` class would be created with the following line:

```
Triangle t = new Triangle();
```

The expression `new Triangle()` makes a call to the default constructor of the `Triangle` class. However, notice that in this scenario, all of the member variables of the `Triangle` object `t` are given a value of 0. Since each member variable of the `Triangle` class represents one of the internal angles of a triangle, this would create a `Triangle` object in an invalid state. This is because the internal angles of a triangle must add up to 180 degrees. So, one could explicitly write a constructor which ensures that anytime a `Triangle` object is created, its angles add up to 180 degrees.

```
class Triangle {
    // the constructor
    Triangle(int a1, int a2, int a3) throws Exception {
        if(a1 < 1 || a2 < 1 || a3 < 1)
            throw new Exception("Angles must be positive!");

        int total = a1 + a2 + a3;
        if(total != 180)
            throw new Exception("Invalid angles!");

        angle1 = a1;
        angle2 = a2;
        angle3 = a3;
    }

    int angle1, angle2, angle3;
}
```

As you can see, the constructor contains logic that validates the angle sizes given to it. That is to say, if a set of given angles does not add up to 180 degrees, it is considered invalid and the constructor of the `Triangle` object returns to the calling code prematurely by throwing an `Exception` which relays the message that the angles are invalid. This forces every `Triangle` object to have angles adding up to 180 degrees. Consider the following statements:

```
Triangle t = new Triangle(30, 60, 90);    // valid Triangle
Triangle t = new Triangle(31, 60, 90);    // invalid Triangle, throws Exception
```

In the first line, a `Triangle` object is created. In the second line, an `Exception` is thrown and the creation of the object is aborted.

Pay attention to the syntax of how a constructor is written: *it has no return value and has the same name as that of the class in which it resides.* Also, note that the following line becomes invalid when a constructor is explicitly defined:

```
Triangle t = new Triangle();      // invalid, no default constructor
```

This is because a default constructor is only provided if no constructor is written in the definition of the class. However, this can be alleviated by creating a ***no-argument constructor***:

```
class Triangle {
    // no-argument constructor
    Triangle() { }
     ...
}
```

In this case, since no code is written into the no-argument constructor, it is problematic as it allows the creation of invalid `Triangle` objects. However, it could be written to create an equilateral triangle like so:

```
Triangle() {
    angle1 = angle2 = angle3 = 60;
}
```

This would then reinstate the restriction that all `Triangle` objects must have angles adding to 180 degrees.

`this` keyword

`this` is a keyword in Java that refers to the ***implicit object***. It is useful in a number of situations and can help to relieve ambiguity and confusion in your code. Regarding the `Triangle` class defined previously, consider the signature of the constructor:

```
Triangle(int a1, int a2, int a3)
```

If one were to name the parameters `angle1`, `angle2`, and `angle3` there would be an ambiguity in the code as the member variables have the same name. In such a scenario, Java hides the member variables. The identifiers `angle1`, `angle2`, and `angle3` are only accessible as the names of the parameters. However, one could write the same constructor like so:

```
Triangle(int angle1, int angle2, int angle3) throws Exception {
    if(angle1 < 1 || angle2 < 1 || angle3 < 1)
        throw new Exception("Angles must be positive!");

    int total = angle1 + angle2 + angle3;
    if(total != 180)
        throw new Exception("Invalid angles!");

    this.angle1 = angle1;
    this.angle2 = angle2;
    this.angle3 = angle3;
}
```

The following lines assign the values of the arguments to the member variables:

```
this.angle1 = angle1;
this.angle2 = angle2;
this.angle3 = angle3;
```

This effectively resolves the ambiguity and provides a way to unmask the member variables. A similar functionality can be seen when using `this` to call another constructor within the same class:

```
Triangle(int angle) throws Exception {
    this(angle, angle, angle);
}
```

The code above allows the specification of a single angle at creation and then passes that angle into the 3-parameter constructor. Of course, there is only one value that will create a valid `Triangle` object with this constructor – namely, 60 degrees – so, this is not a terribly useful constructor, but the point is to illustrate that a constructor can be called from within another constructor. Notice that the constructor signature is followed by the phrase `throws Exception`. This is because the following line may throw an `Exception`:

```
this(angle, angle, angle);
```

A call to any method that throws an `Exception` may cause the calling function to throw an `Exception`.

`static` keyword

`static` is a keyword in Java that can modify the defintion of a class, a method, or a member variable. For the time being, we will only be concerned with methods and member variables. `static` should be familiar from the definition of any `main()` method. As we know, member variables are specific to a particular object which is an instance of a class. Additionally, member methods can only be called through a particular object. Consider the `Triangle` class again:

```
Triangle t1 = new Triangle(30, 60, 90);
Triangle t2 = new Triangle(20, 50, 110);
```

The `Triangle` objects `t1` and `t2` both have three `int` variables, but values of those variables are different. Occasionally, we will want to have variables that belong to *all members of a class*. That is, some variables should be shared by all objects and should exist before any objects have been created. Furthermore, some methods need to be called before the creation of any objects of that class – i.e. `main()`. For all of these scenarios, we use the keyword `static`.

Consider the following class:

```
class Circle {
    Circle(double radius) {
        this.radius = radius;
    }

    double area() {
        double area = PI * radius * radius;
        return area;
    }

    double radius;
    static double PI = 3.14159;
}
```

In this scenario, the variable `PI` has been declared `static`. The benefit of a `static` variable in this case is that we only need one space to store the value of `PI`. There is no reason to have the same variable in every single object because the value is the same for all objects. However, as the code is currently written, the value of `PI` could be changed by any object. For instance, the following code shows that a change in one object produces a change in another object:

```
System.out.println(Circle.PI);    // prints 3.14159

Circle c1 = new Circle(3);
c1.PI = 10;
System.out.println(Circle.PI);    // prints 10

Circle c2 = new Circle(4);
System.out.println(c2.PI);        // prints 10
```

In the case of the variable `PI`, it would be unwise to allow this to change as it is really a constant value. Changes can be prevented by the use of the `final` modifier which, in essence, creates a read-only variable:

```
final static double PI = 3.14159;
```

If `PI` is declared as it is directly above, then the following lines, or any other that try to write to `PI`, will not compile:

```
// these are all compiler errors!
Circle.PI = 10;
c1.PI = 10;
c2.PI = 10;
```

Also, notice how `PI` can be accessed even before any objects have been created:

```
System.out.println(Circle.PI);    // prints 3.14159
```

As mentioned before, this is because `static` variables and methods are *are not dependent on any particular object of the class.*

One practical use of the `static` keyword can be illustrated with a conventional deisgn pattern called a *static factory method*. This technique encapsulates all the logic of validating the data fields into a `static` method which then returns an instance of the class. This can be beneficial for a few reasons, two of which are discussed in the following paragraphs.

1. Static factory methods provide a way of side-stepping an inability to have two constructors with identical parameter lists. For instance, in the case of our `Triangle` class we may want to write a constructor that takes three `int` arguments that represent the lengths of the three sides:

```
class Triangle {
    // a second constructor -- this will not compile!
    Triangle(int side1, int side2, int side3} throws Exception {
        if(side1 < 1 || side2 < 1 || side3 < 1)
            throw new Exception("Sides must be positive!");

        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
        calculateAngles();
    }

    Triangle(int angle1, int angle2, int angle3) {
        ...
    }

    void calculateAngles() {
        // calculates the angles given side lengths
    }

      ...

    int side1, side2, side3;
}
```

The code above will not compile because the two constructors have identical parameter lists. When creating a `Triangle` object it would be ambiguous which constructor was intended:

```
Triangle t = new Triangle(20, 87, 73);    // which constructor is called?
```

One might think that the solution would be to have one constructor take `double`'s and the other `int`'s. This is not a satisfying solution, as both the angles and sides of a triangle could be decimal or integer values depending on what is relevant to the class. Really, any variation in numeric type would not be a good way to distinguish the two constructors as then the programmer using the class would also have to remember which constructor corresponds to that particular parameter list. The best way to deal with this is to create some static factory methods.

```
    class Triangle {
        private Triangle() { }

        static Triangle angles(int angle1, int angle2, int angle3) throws Exception {
             if(angle1 < 1 || angle2 < 1 || angle3 < 1)
                throw new Exception("Angles must be positive!");

            int total = angle1 + angle2 + angle3;
            if(total != 180)
                throw new Exception("Invalid angles!");

            Triangle t = new Triangle();
            t.angle1 = angle1;
            t.angle2 = angle2;
            t.angle3 = angle3;
            t.calculateSides();
            return t;
        }

        static Triangle sides(int angle1, int angle2, int angle3) throws Exception {
            if(side1 < 1 || side2 < 1 || side3 < 1)
                throw new Exception("Sides must be positive!");

            Triangle t = new Triangle();
            this.side1 = side1;
            this.side2 = side2;
            this.side3 = side3;
            calculateAngles();
            return t;
        }

            ...
    }
```

As can be seen, the factory methods verify that all the data is correct and then create a valid `Triangle` object with that data. Notice that the constructor is now `private`. This is so `Triangle` objects can *only be created through the factory method.* This guarantees that all of our `Triangle` objects are valid.

```
    Triangle t1 = Triangle.angles(30, 60, 90);
    Triangle t2 = Triangle.sides(13, 14, 15);
    Triangle t3 = new Triangle();               // this will not compile!
```

Notices also that the static factory methods have descriptive names, which makes them much easier to use. A constructor is identified purely by its parameter list as it cannot have any other name besides that of its class.

2. Static factory methods are not required to create an object when they are invoked. The original `Triangle` constructor threw an `Exception` if the arguments it received were invalid. This can be a serious design flaw in certain scenarios. The problem is: the moment the constructor is called, memory is dynamically allocated to create the object whether or not the data is valid. So, if the constructor throws an `Exception`, the memory is allocated but then not used.

Consider the following:

```
Triangle t = null;
try {
    t = new Triangle(31, 60, 90);    // invalid args to original constructor
} catch(Exception e) {
    System.err.println(e.getMessage());
}

System.out.println(t);              // prints null
```

The constructor allocates memory for the new `Triangle` object, but since an `Exception` is thrown the object never actually gets assigned to `t`. In other words, there is no way to stop the construction process and immediately free the memory, nor is it accessible after it is created. The invalid object will continue to occupy memory and remain unusable until the garbage collector comes around. Of course, this is different with our static factory methods:

```
Triangle t = null;
try {
    t = Triangle.angles(31, 60, 90);    // invalid args to original constructor
} catch(Exception e) {
    System.err.println(e.getMessage());
}

System.out.println(t);              // prints null
```

In this scenario, `t` still contains the value `null` by the last line; however, no memory is dynamically allocated to create a `Triangle` object. This is because the validation in the static factory method happens *before the call to the constructor!*